浙江大学

本科实验报告

课程名称：计算机体系结构

姓　　名：范源颢

学　　院：计算机科学与技术学院

　　系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：3180103574

指导教师：陈文智

2020 年　12 月 22　日

浙江大学实验报告

课程名称：计算机体系结构 实验类型：综合

实验项目名称：多周期访存的流水线 CPU

学生姓名：范源颢 专业：计算机科学技术 学号：3180103574

同组学生姓名：周寒靖 指导老师：陈文智

实验地点：曹光彪西楼-301 实验日期：2020 年 1 月 3 日

一、 实验目的和要求

目的：

1. 理解 CPU 多周期访存的原理以及多周期访存对 CPU 的影响
2. 掌握多周期访存流水线 CPU 的设计方法
3. 掌握多周期访存流水线 CPU 的程序验证方法

要求：

1. 设计流水线 CPU，使得它支持 2 条中断指令，重新设计指令寄存器和数据寄存器（Inst_rom & data_ram）,完善 CPU 控制器（CPU Controller）和数据通路（datapath）。同时增加一个协处理器寄存器记录运行周期数
2. 用程序验证单周期 CPU，观察程序的执行情况。

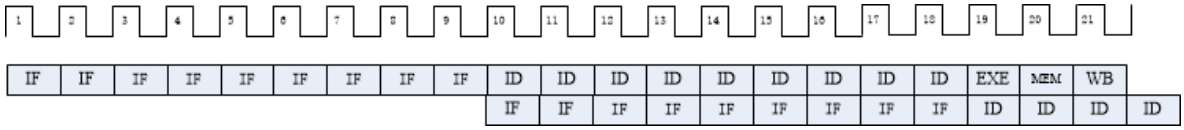
二、 实验内容和原理

2.1 多周期访存

我们知道，在现实的计算机体系结构中，CPU 运行的速度远远快于指令寄存器和数据寄存器的访存速度，但是在我们的前几次实验中，这个过程没有体现，原因可能是因为我们的时钟周期设置的太慢了，或者内存的大小设置的太小了，导致访存基本上都能在一个时钟周期内完成，为了模拟现实世界中 CPU 的运行情况，我们认为将这个特性添加上去，主要需要实现的是：在访存的过程中，流水线需要

主动停顿 8 周期，这样一来，指令访存和数据访存都需要 9 个周期完成。

如下图所示：



2.2 指令寄存器和数据寄存器的工作任务

指令寄存器和数据寄存器额处理逻辑比较简单和类似，这里就一起说了，首先，我们要维护一个计数器寄存器 counter，当我们发现地址线有索引 addr 信号时，我们判断此时钟周期的 addr 和上个周期的信号是否相同（我们在停顿的过程中，addr 信号可能会一直从 CPU 中送来，为此我们需要依次判定我们是否在停顿状态，当然，为了保存上一次的 addr 信号，我们需要一个 addr_prev 寄存器）。如果相同，说明我们处在等待状态，那么我们将利用 counter 寄存器计数，一旦寄存器已经为 8，那么我们需要将数据读出（out <= data[addr[ADDR_WIDTH - 1 : 0];）之后给出一个 ack 信号，表示我们已经完成了数据的读出。假如这次的 addr 和上次的 addr（addr_prev）不同，那么就需要将 counter 和 ack 都置位为 0。当然在周期末尾，不要忘记将 addr_prev 赋值为 addr.

我们的寄存器需要发出的信号时 rom_stall 信号，这个信号的定义应当是：

stall = cs & ~ack,

cs 表示 chip select，表示片选信号，数据通路选择了这个芯片（inst_rom / data_ram）,~ack 则表示这个芯片的读取工作还没有完成，逻辑上，被选择需要执行的芯片假如没有完成这个工作，那么久应该发出停顿信号。存储器需要维护的是 ack 信号，其维护仰赖于 addr_prev 和 addr 的比较，我们因此希望在流水线停顿时，流水线不继续执行，而是停留在读取阶段，不断的给出相同的 addr 信号。

2.3 存储器停顿信号的处理

我们首先考虑 inst_rom 的停顿，我们需要将 stall 信号发送给 controller 处理，controller 集中处理 stall，如果直接发送给 datapath 可能会出现功能混淆的情况，假如出现 rom_stall 的情况，说明我们应该延长 IF 阶段和 ID 阶段，也就是将 if_en 和 id_en 都置 0，这样一来，inst_addr 就不再更新了，只能发送上个周期的 inst_addr 信号。（因为 inst_addr 是一个寄存器输出）于是就实现了停顿的交互。

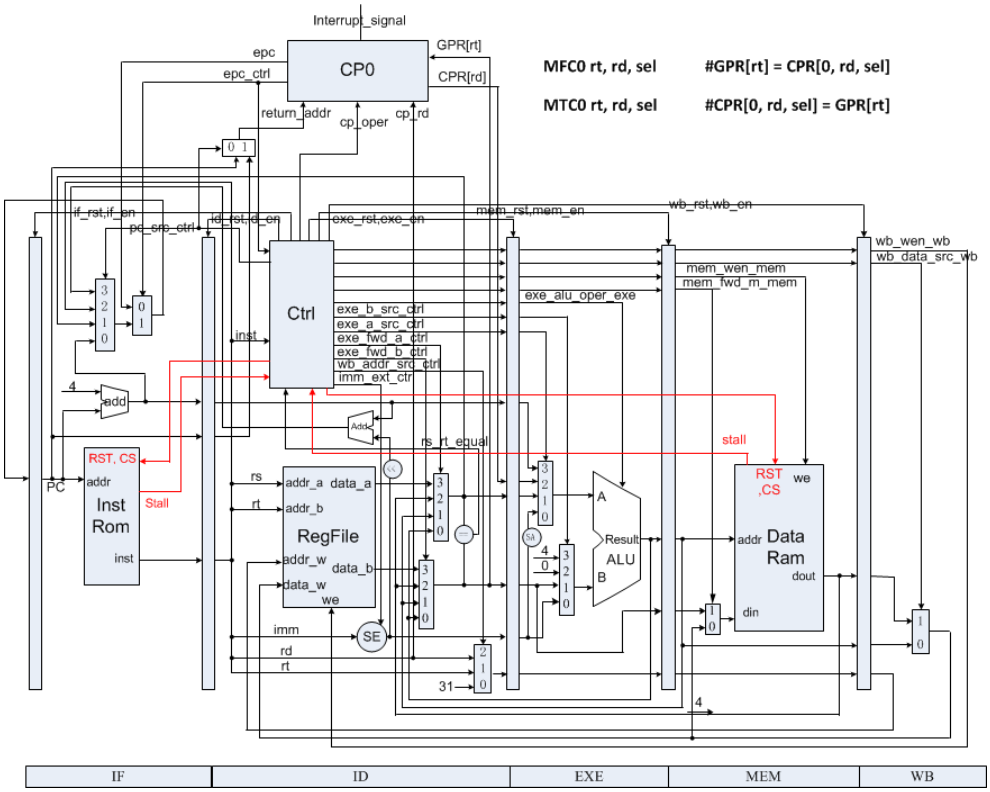
但是，我们还要考虑 EXE 阶段的指令，我们不能也让 EXE 阶段的命令重复执行，因此我们不能选择设置 exe_en 为 0，否则，上个周期的 EXE 阶段的指令的输入依旧会在这个周期输出（xxx_en 只是控制 xxx 阶段的输入是否会更新而已，不会真的控制元件是否进行运算），为此，我们需要将 exe_rst 置 1，使得 EXE 阶段的输入全部更新为 0，那样，我们只有在九个周期之后，IF，ID 阶段的新的输出出来之后，在开始流水线，而一旦 exe_rst 置 1，之后的流水线控制信号都不需要再做处理，因为自 EXE 之后的那些阶段得到的输入都是 0，也就都在执行空操作。

之后，data_ram 的停顿，和 inst_rom 的停顿都是一样的，区别在于，我们需要将 if、id、exe、mem 阶段的 enable 信号都置 0，使得他们停顿（因为 data_ram 的停顿信号时在 mem 阶段产生的），而 mem 阶段之后的 wb 阶段，我们要将他的 rst 信号置 1.

2.4 增加 CP0 的 TCR 功能

我们的 CP0TCR 其实不参与指令停顿的实现过程，他只是用于记录程序执行的时钟周期数目而已，TCR 的意义是时钟节拍计数寄存器（Tick Counter Register），一旦 CPU 经过了一个时钟周期，我们就会给这个寄存器的值自增。实际情况下，我们会用 MFC0 读出 TCR 的值。在本次实验中，我们会用 CP0_TCR = 9 这个宏来描述，表示 CP0 的第九号寄存器是 TCR。

2.5 实验参考电路图：



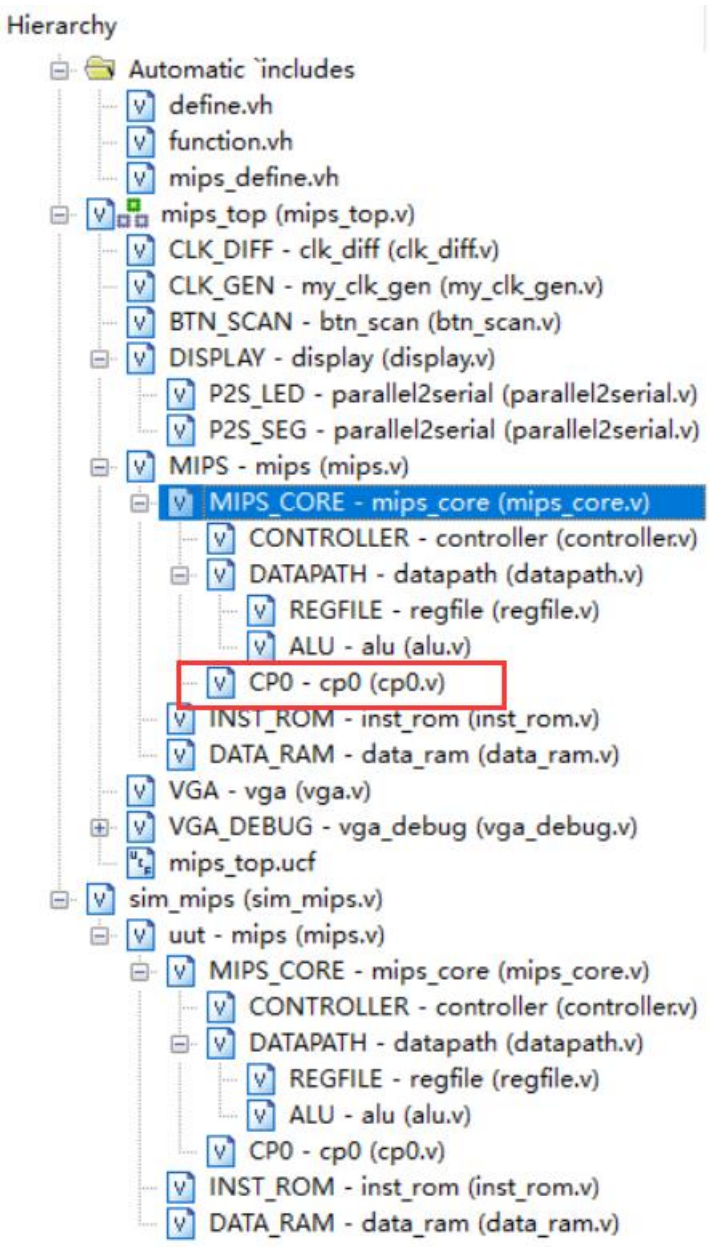
主要是增加了 CS 信号和 Stall 信号，其内容我们已经阐述过了，这里就不再赘述

2.6 我们的主要工作：

1. 处理 CPU Controller，添加 rom_stall 和 ram_stall 等信号的解释
2. 处理 inst_rom 和 data_ram 模块，实现多周期访存
3. 新增 CP0 模块的 TCR 功能
4. 处理 mips_core.v，将 Controller 的输出和 CP0，datapath 的输入连接起来（datapath 倒不需要做什么改动）
5. 对于新增信号的判断，我们如果想要引入宏增强可读性，那么我们需要修改 mips_define.vh
6. 我们需要处理 data_mem.hex 和 inst_mem.hex，修改测试程序，使得我们的新的指令能被测试到。

三、 实验过程和数据记录

实验的文件结构如下：



Automatic includes 文件夹下定义了我们可以在代码中使用的头文件；

mips_top.v 是实验的主要工程，实现了对于 MIPS 指令的模拟，clk_diff 和 clk_gen 模块用于 CPU 内置时钟的信号生成的处理，便于各个进程的同步。Btn_san 是处理输入信号的模块，时限完成的 mips_top 模块将烧录到实验室的 sword 开发板上，通过按钮的控制反映 CPU 的计算情况，信息通过 3 个模块输出，分别是 DISPLAY 模块和 VGA，VGA_DEBUG 模块，mips_top.ucf 规定了电路的引脚，使得烧录之后的程序可以嵌入开发板；

sim_mi.v 实际上是一个测试用的工程文件，用于在开发者的电脑上，用 Xilinx ISE 仿真烧录之后的输入输出过程，uut 文件指定了我们的模拟方式（通过内置时钟不断读取 inst_rom 和 data_ram 中的信息，并且把相关的信号反馈到电路图上），仿真检测的对象自然是 mips_core，这个 mips_core 核上文中 mips_top 模块中的 mips_core 是同一个文件。

我们需要完成 mips_core 中的代码补全。mips.v 文件划分为 MIPS_CORE 和 inst_rom，data_ram，完成了二者（CPU 和内存）的交互，mips_core.v 划分为 controller.v 和 datapath.v，完成了二者（控制器和数据通路）的交互，我们重点需要完成 controller.v 和 datapath.v 的代码补全。同时注意 cp0 的代码

1. inst_rom 和 data_ram：

我们以 inst_rom 为例，展示控制周期停顿的代码：

```
always @(negedge clk) begin
    if(rst)begin
        counter=0;
        ack=0;
    end
    else begin
        if(addr_previous==addr)begin
            counter=counter+1;
            if(counter==8)begin
                out<=data[addr[ADDR_WIDTH-1:0]];
                ack=1;
            end
        end
        else begin
            counter=0;
            ack=0;
```

```
            end
            addr_previous=addr;
        end
    end
    assign rom_stall=cs&~ack;
```

data_ram 也类似，如 2.2 所述，这样可以完成停顿周期为 8 的任务

2. CPU Controller，mips_core 和 mips

我们对于 controller 的内容修改如下，我们需要给出 cs 信号，同时解释 data_ram 和 inst_rom 发出的 stall 信号。

我们在程序开始时中，首先初始化 rom_cs 为 1，因为在 controller 工作的 id 阶段，上一条指令在 if 我们必然已经片选了 inst_rom 希望执行指令取出。

而对于 ram_cs，我们在初始化的时候会将他们置为 0，只有 id 的结果有 lw, sw 其中之一时，我们才将他们置 1，只有载入字和存储字两条指令才会涉及到数据存储器的片选。

```
            INST_LW: begin
                imm_ext = 1;//
                exe_b_src = EXE_B_IMM;//
                exe_alu_oper = EXE_ALU_ADD;//
                mem_ren = 1;//
                wb_addr_src = WB_ADDR_RT;//
                wb_data_src = WB_DATA_MEM;//
                wb_wen = 1;//
                rs_used = 1;//
                is_load = 1; //
                //added in exp7-------
                ram_cs = 1;
            end
            INST_SW: begin
                imm_ext = 1;//
                exe_b_src = EXE_B_IMM;//
                exe_alu_oper = EXE_ALU_ADD;//
                mem_wen = 1;//
                rs_used = 1;//
                rt_used = 1;//
                is_store = 1;//
                //added in exp7 -----
                ram_cs = 1;
            end
```

余下的就是 rom_stall 和 ram_stall 的处理，原理如 2.3 所述，代码如下：

```
        else if(rom_stall) begin
```

```
            if_en = 0;
            id_en = 0;
            exe_rst = 1;
        end
        else if (ram_stall) begin
            if_en = 0;
            id_en = 0;
            exe_en = 0;
            mem_en = 0;
            wb_rst = 1;
        end
```

在 mips_core.v 中，我们注意要加上 rom/ram_stall 和 rom/ram_cs 信号，作为 controller 的接口，然后在 mips.v 中，将这些接口和 inst_rom，inst_ram 接和起来。

```
    // mips core
    mips_core MIPS_CORE (
        ......

        //exp7 added port
        .ram_stall(ram_stall),
        .rom_stall(rom_stall),
        .ram_cs(ram_cs),
        .rom_cs(rom_cs)

        );

    inst_rom INST_ROM (
        .clk(clk),
        .addr({2'b0, inst_addr[31:2]}),
        //.addr(inst_addr),
        .dout(inst_data),

        //exp7 added
        .rom_stall(rom_stall),
        .cs(rom_cs),
        .rst(rst)
        );

    data_ram DATA_RAM (
        .clk(clk),
        .we(mem_wen),
        .addr({2'b0, mem_addr[31:2]}),
        //.addr(mem_addr),
        .din(mem_data_w),
```

```
        .dout(mem_data_r),

        //exp7 added
        .ram_stall(ram_stall),
        .cs(ram_cs),
        .rst(rst)
        );
```

3. cp0 和 mips_define：

我们只需要在每个时钟上升沿增加一次 TCR 寄存器的值就可以了

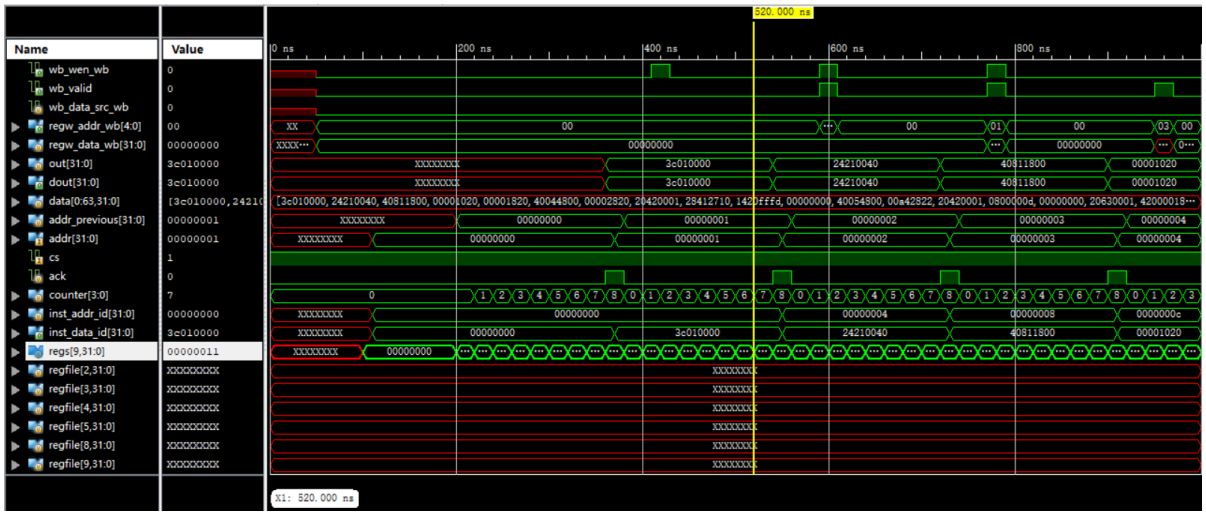
```
        if(rst)begin
            regs[CP0_TCR]=0;
        end
        else begin
            regs[CP0_TCR]=regs[CP0_TCR]+1;
        end
```

不要忘记在头文件中添加都 CP0_TCR 的定义：

```
localparam
    //CP0_SR = 0,
    //CP0_EAR = 1,
    CP0_EPCR = 2,
    CP0_EHBR = 3,
    //CP0_IER = 4,
    //CP0_ICR = 5,
    //CP0_PDBR = 6,
    //CP0_TIR = 7,
    //CP0_WDR = 8;
     CP0_TCR = 9;////
```

四、 实验结果分析

4.1 仿真结果：



我们在图片右侧的紫框中找到相应的芯片元件，拖入左侧的图中就可以得到输入输出的信号波形图，注意设定仿真时长，并且显示方式为 16 进制(便于阅读)。

主要观察的波形是 regfile 和 inst 信号的输出，通过观察 inst 信号我们可以确定跳转的指令是否正确，通过观察 regfile 我们可以确定写回的信号是否正确。

4.2 测试用程序介绍

我们测试用的程序如下（注意和之前的程序不同）：

```
 0: 3c010000    lui     R1,0x0      # main entry
 4: 24210040    addiu   R1,R1,64
 8: 40811800    mtc0    R1,R3
 c: 00001020    add     R2,R0,R0
10: 00001820    add     R3,R0,R0
14: 40044800    mfc0    R4,R9
18: 00002820    add     R5,R0,$0
1c: 20420001    addi    R2,R2,1     # loop
20: 28412710    slti    R1,R2,10000
24: 1420fffd    bne     $1,R0,-2    # jump to loop
28: 00000000    nop
2c: 40054800    mfc0    R5, R9
30: 00a42822    sub     R5, R5, R4
34: 20420001    addi    R2,R2,1     # loop
38: 0800000d    j       34
3c: 00000000    nop
40: 20630001    addi    R3,R3,1     # handler
44: 42000018    eret
48: 00000000    nop
```

最右侧的 16 进制数是机器码，直接存放在 inst_rom.hex 中，被 CPU 读取。左侧的内容是机器码翻译之后的汇编语言，以及汇编语言的语义（用注释表示）

本次实验不涉及到内存，所以我们不涉及 data_rom.hex，事实上，这个文件里面的内容可以是空的，也尅是任意的其他内容。

上述汇编语言的结果如下：首先，我们使用 lui 指令，将 R1 用 0 填充，再将 64 用 addiu 指令填入 R1（十六进制表现为 40），之后，我们把 R1 中的值移入到 CP0 的 R3 寄存器（EHBR），之后，我们将 R2、R3 置 0，从 R4 中读出 CP0 的 R9（TCR），这个时候的 R9 按照时钟周期计数应该是 7 * 9 = 63 = 0x3f.

之后我们将 R5 置为 0，开始反复 R2 加 1 的操作，然后判断 R2 是否小于 10000，

若是，则 R1 置 1，再判断 R1 是否和 0 相等，若不等，则上跳两条指令，R2 继续加 1，这实际上是 R2 自增 10000 的操作，完成自增之后，跳转到 2c, 利用指令 mfc0 将 TCR 的值读取出来，观察经过了多少周期（应该经过了 10000 * 7 = 0x57e91 个），之后在讲 R5 和 R4（0x3f）相减，结果应当为 0x57e52，值存入 R5 中。

最后，我们进行 R2 反复加 1 的死循环结束程序。另外，如果有中断，那么中断响应程序（自 40 开始），和上一次一样，R3 加 1 之后，返回中断处继续执行。

3.3 仿真程序和测试结果：

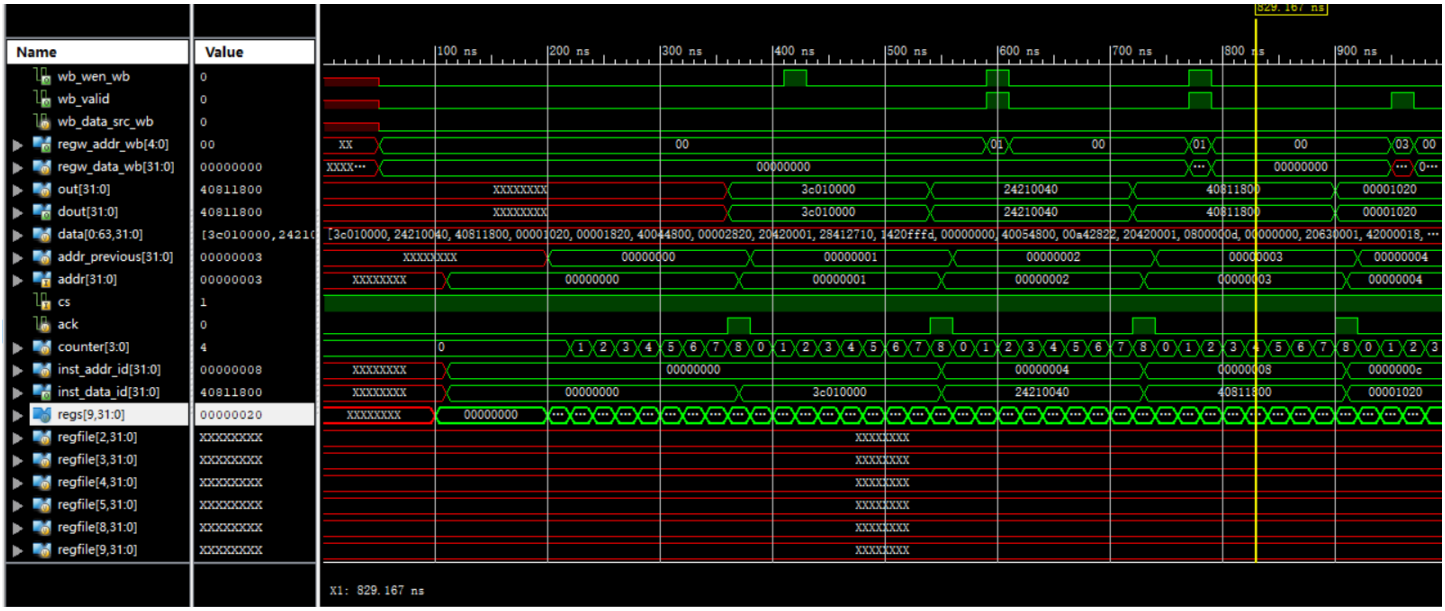
我们的仿真代码如下：

```
`timescale 1ns / 1ps


module sim_mips;

    // Inputs
    reg debug_en;
    reg debug_step;
    reg [6:0] debug_addr;
    reg clk;
    reg rst;
    reg interrupter;

    // Outputs
    wire [31:0] debug_data;

    // Instantiate the Unit Under Test (UUT)
    mips uut (
        .debug_en(debug_en),
        .debug_step(debug_step),
        .debug_addr(debug_addr),
        .debug_data(debug_data),
        .clk(clk),
        .rst(rst),
        .interrupter(interrupter)
    );

    initial begin
        // Initialize Inputs
        debug_en = 0;
        debug_step = 0;
        debug_addr = 0;
```

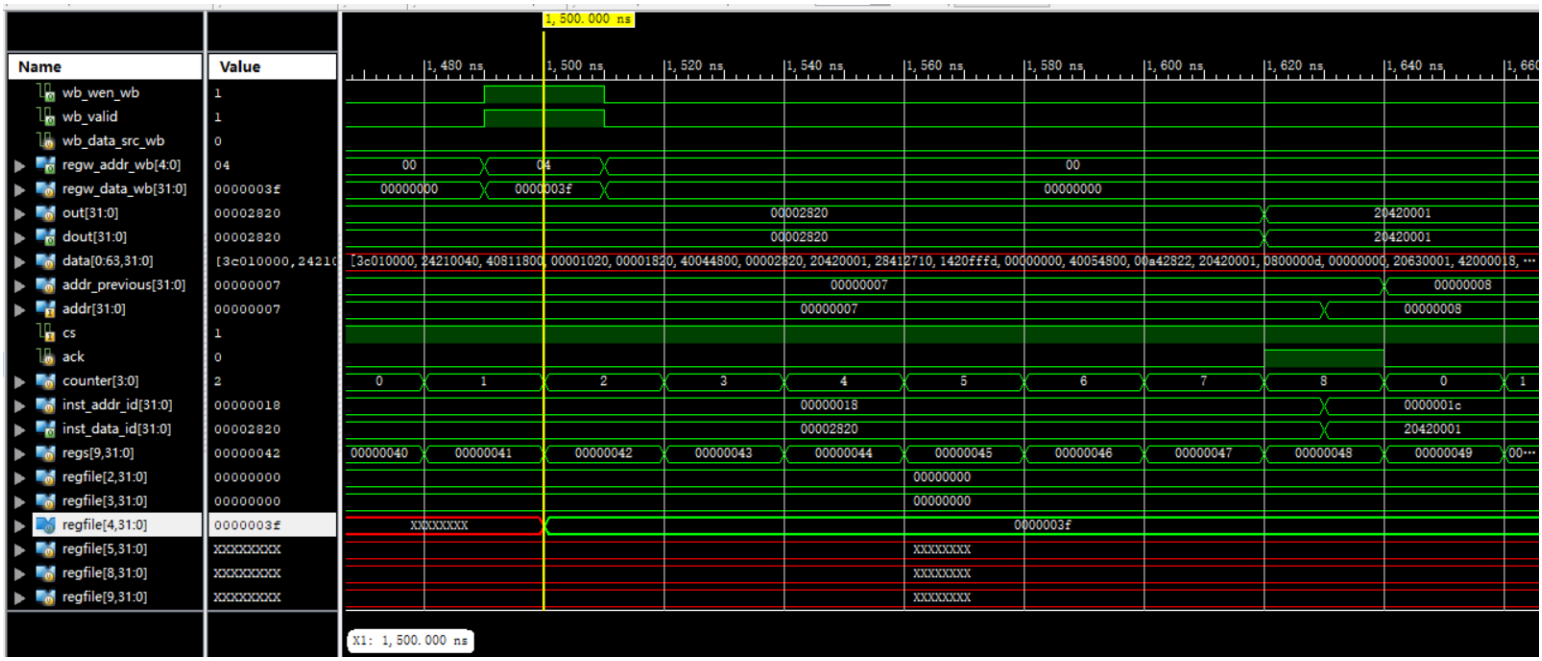
```
        clk = 0;
        rst = 0;
        interrupter = 0;

        #100 rst = 1;
        #100 rst = 0;

        //#27000; interrupter = 1;//exp7 调试的断点要晚一点
        //#40; interrupter = 0;

        //#40 interrupter = 1;
        //#40 interrupter = 0;

        #180;
    end
    initial forever #10 clk = ~clk;

endmodule
```

我们首先不设置中断，观察实验情况。

开始阶段：

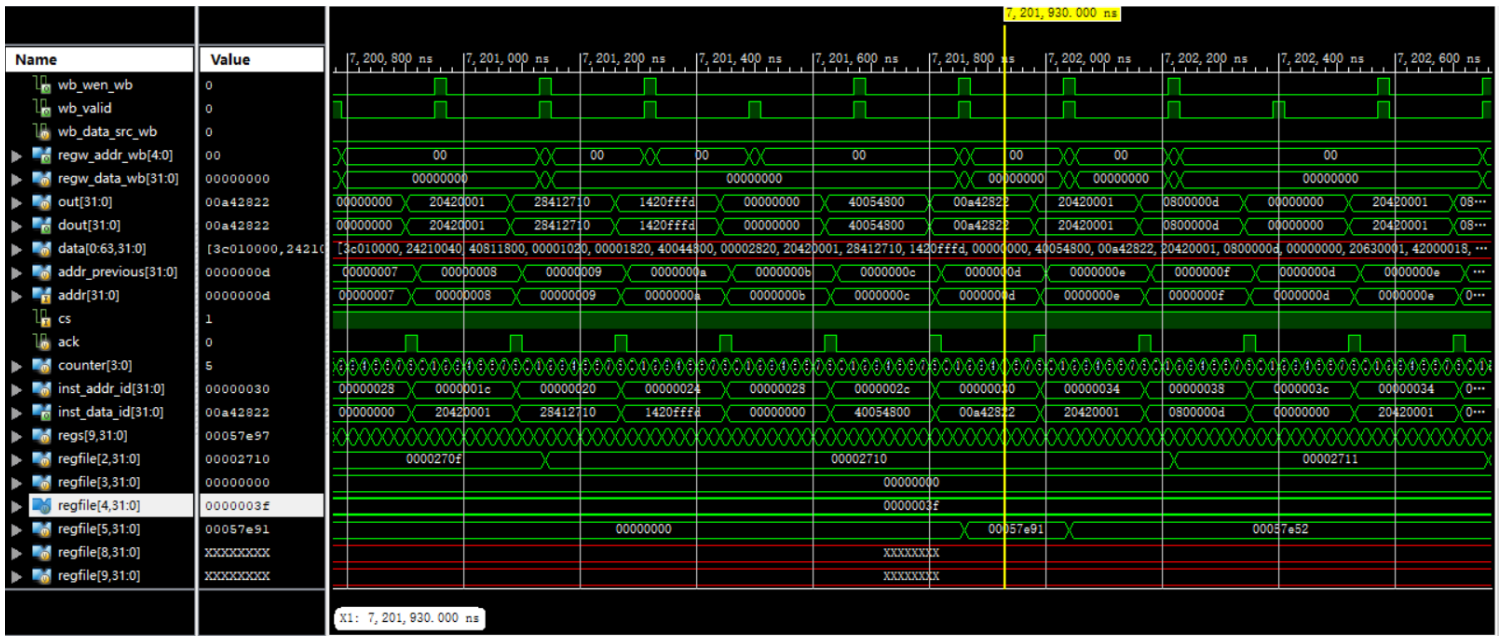


（第一次 mfc0 部分）



（第二次 mfc0 部分）

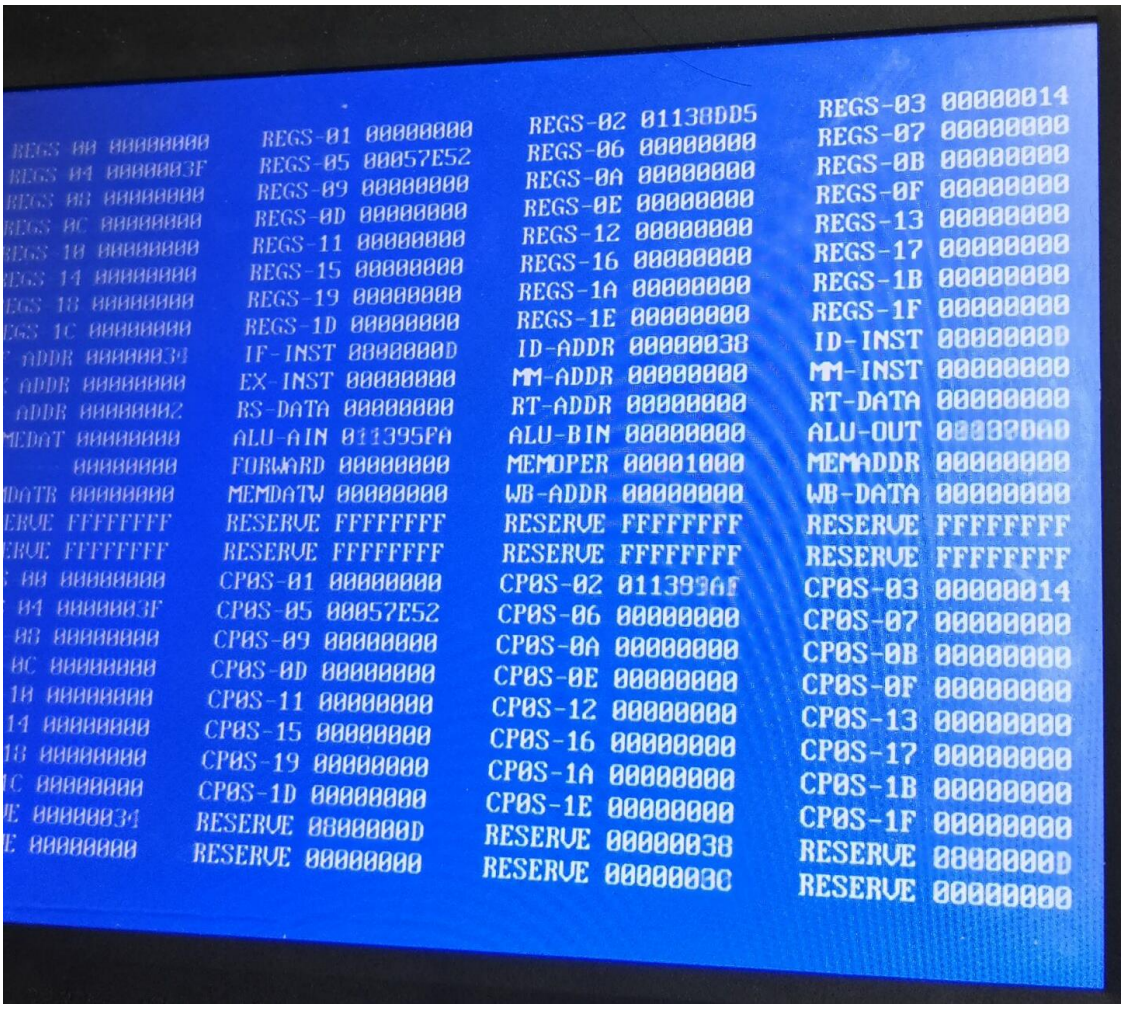
（这次需要等待的时钟周期比较长，一般在 7,200000ns 才会有 R5 的变化，毕竟循环了 10000 次）

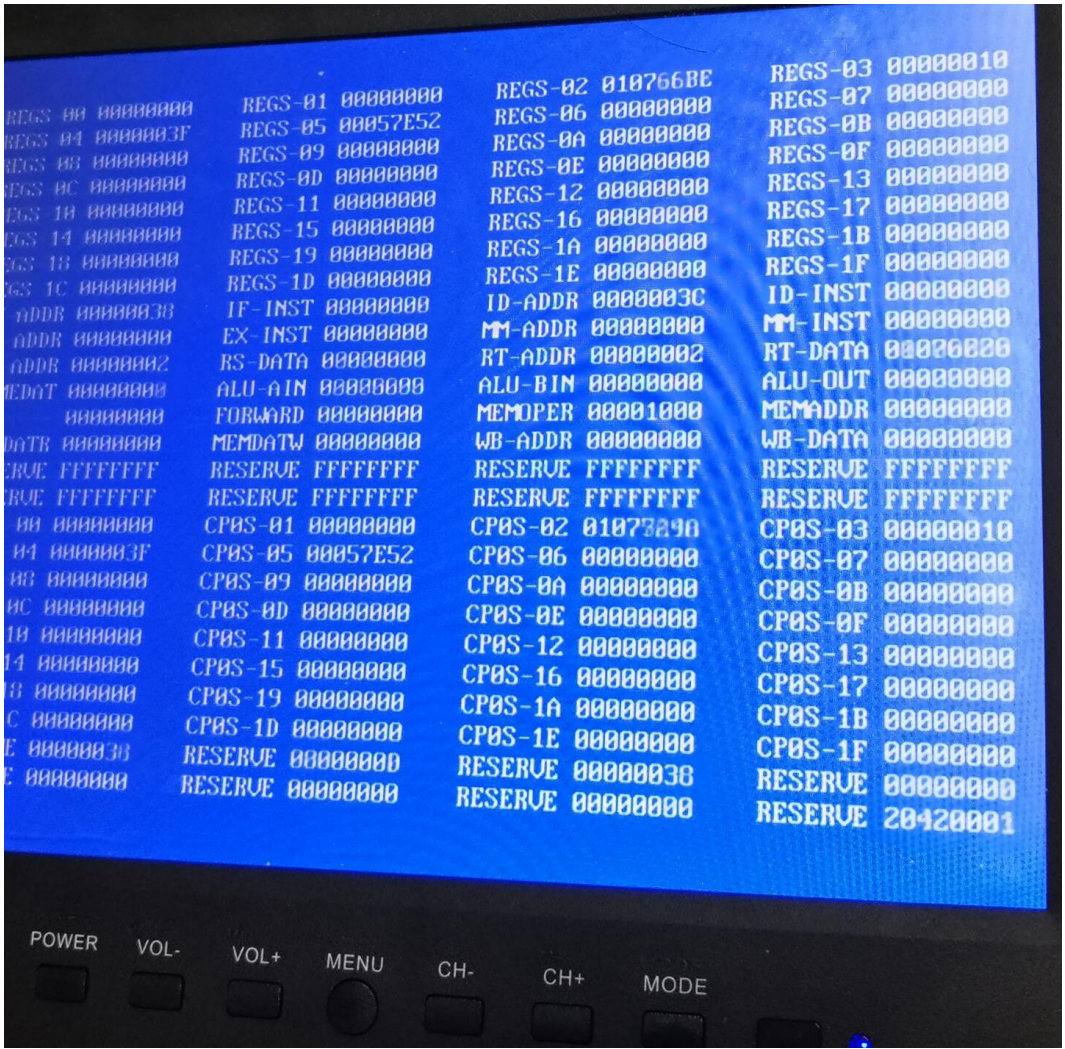


4.4 在 sword 开发板上的仿真结果

因为已经通过现场测试，这里就不再展示全部的情况，仅仅选择部分展示

下文节选一些中间过程(测试了中断)


REGS-01 00000000
REGS-05 00057E52
REGS-09 00000000
REGS-0D 00000000
REGS-11 00000000
REGS-15 00000000
REGS-19 00000000
REGS-1D 00000000
IF-INST 0000000D
EX-INST 00000000
RS-DATA 00000000
ALU-AIN 011395FA
FORWARD 00000000
MEMDATW 00000000
RESERVE FFFFFFFF
RESERVE FFFFFFFF
CP0S-01 00000000
CP0S-05 00057E52
CP0S-09 00000000
CP0S-0D 00000000
CP0S-11 00000000
CP0S-15 00000000
CP0S-19 00000000
CP0S-1D 00000000
RESERVE 0000000D
RESERVE 00000000
REGS-02 01138DD5
REGS-06 00000000
REGS-0A 00000000
REGS-0E 00000000
REGS-12 00000000
REGS-16 00000000
REGS-1A 00000000
REGS-1E 00000000
ID-ADDR 00000038
MM-ADDR 00000000
RT-ADDR 00000000
ALU-BIN 00000000
MEMOPER 00001000
WB-ADDR 00000000
RESERVE FFFFFFFF
RESERVE FFFFFFFF
CP0S-02 01139961
CP0S-06 00000000
CP0S-0A 00000000
CP0S-0E 00000000
CP0S-12 00000000
CP0S-16 00000000
CP0S-1A 00000000
CP0S-1E 00000000
RESERVE 00000038
RESERVE 0000003C
REGS-03 00000014
REGS-07 00000000
REGS-0B 00000000
REGS-0F 00000000
REGS-13 00000000
REGS-17 00000000
REGS-1B 00000000
REGS-1F 00000000
ID-INST 0000000D
MM-INST 00000000
RT-DATA 00000000
ALU-OUT 00000000
MEMADDR 00000000
WB-DATA 00000000
RESERVE FFFFFFFF
RESERVE FFFFFFFF
CP0S-03 00000014
CP0S-07 00000000
CP0S-0B 00000000
CP0S-0F 00000000
CP0S-13 00000000
CP0S-17 00000000
CP0S-1B 00000000
CP0S-1F 00000000
RESERVE 0000000D
RESERVE 00000000


REGS-01 00000000
REGS-05 00057E52
REGS-09 00000000
REGS-0D 00000000
REGS-11 00000000
REGS-15 00000000
REGS-19 00000000
REGS-1D 00000000
IF-INST 00000000
EX-INST 00000000
RS-DATA 00000000
ALU-AIN 00000000
FORWARD 00000000
MEMDATW 00000000
RESERVE FFFFFFFF
RESERVE FFFFFFFF
CP0S-01 00000000
CP0S-05 00057E52
CP0S-09 00000000
CP0S-0D 00000000
CP0S-11 00000000
CP0S-15 00000000
CP0S-19 00000000
CP0S-1D 00000000
RESERVE 0000000D
RESERVE 00000000
REGS-02 010766BE
REGS-06 00000000
REGS-0A 00000000
REGS-0E 00000000
REGS-12 00000000
REGS-16 00000000
REGS-1A 00000000
REGS-1E 00000000
ID-ADDR 0000003C
MM-ADDR 00000000
RT-ADDR 00000002
ALU-BIN 00000000
MEMOPER 00001000
WB-ADDR 00000000
RESERVE FFFFFFFF
RESERVE FFFFFFFF
CP0S-02 010738A0
CP0S-06 00000000
CP0S-0A 00000000
CP0S-0E 00000000
CP0S-12 00000000
CP0S-16 00000000
CP0S-1A 00000000
CP0S-1E 00000000
RESERVE 00000038
RESERVE 00000000
REGS-03 00000010
REGS-07 00000000
REGS-0B 00000000
REGS-0F 00000000
REGS-13 00000000
REGS-17 00000000
REGS-1B 00000000
REGS-1F 00000000
ID-INST 00000000
MM-INST 00000000
RT-DATA 00000000
ALU-OUT 00000000
MEMADDR 00000000
WB-DATA 00000000
RESERVE FFFFFFFF
RESERVE FFFFFFFF
CP0S-03 00000010
CP0S-07 00000000
CP0S-0B 00000000
CP0S-0F 00000000
CP0S-13 00000000
CP0S-17 00000000
CP0S-1B 00000000
CP0S-1F 00000000
RESERVE 00000000
RESERVE 20420001
POWER
VOL-
VOL+
MENU
CH-
CH+
MODE

实验中 mips_top.v 中的语句

```
assign
     btn_step = btn[16],
     btn_interrupt = btn[19];////interrupt button
```

将触发中断的信号绑定在了 19 号按钮上，我们可以按压开发板上的这一按钮实现中断。

五、 讨论与心得

感觉仿真的时候一直没法产生正确的结果，以为是自己程序的问题，结果发现可能是终端信号的问题，中断如果在程序中产生的话，那么也会影响 TCR 的计数，为此仿真的时候我们将中断关闭，得到了正确的结果。